BRICS

Basic Research in Computer Science

BRICS RS-96-38 Hüttel & Kleist: Objects as Mobile Processes

Objects as Mobile Processes

Hans Hüttel
Josva Kleist

BRICS Report Series **RS-96-38**

ISSN 0909-0878 **October 1996**

**See back inner page for a list of recent publications in the BRICS
Report Series. Copies may be obtained by contacting:**

**BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark**

**Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk**

**BRICS publications are in general accessible through World Wide
Web and anonymous FTP:**

`http://www.brics.dk/`
`ftp://ftp.brics.dk/pub/BRICS`

Objects as mobile processes

Hans Hüttel and Josva Kleist*

October 29, 1996

Abstract

The object calculus of Abadi and Cardelli [AC96, AC94b, AC94a] is intended as model of central aspects of object-oriented programming languages. In this paper we encode the object calculus in the asynchronous π-calculus without matching and investigate the properties of our encoding.

1 Introduction

In [AC96, AC94b, AC94a] Abadi and Cardelli investigate several versions of an object oriented calculus (the ς-calculus) with respect to its type system. The primary motivation behind the ς-calculus is to find a simple foundation for object oriented languages, just as the λ-calculus forms a foundation for functional programming languages.

In the ς-calculus the central concept is that of an *object*. Objects are built from object formation ($[l_i = \varsigma(x_i)b_i]$) where we create an object with methods $l_i = \varsigma(x_i)b_i$, method activation ($a.l$) where we activate the method named l in object a, and method override ($a.l \Leftarrow \varsigma(x)b$) where the method named l in object a is exchanged with the new method $l = \varsigma(x)b$. Despite its apparent simplicity, the ς-calculus has previously shown its ability to express several object oriented features within the calculus and it is also possible to encode the λ-calculus[Abr89].

In this paper we shall describe an encoding of the simplest version of the ς-calculus into an asynchronous version of the π-calculus[PW92] and investigate the properties of this encoding. In particular, the encoding is sound under the operational semantics of the ς-calculus. We also show that

*Address: Dep. of Computer Science, Aalborg University, Frederik Bajersvej 7, 9220 Aalborg, Denmark. Email: {hans,kleist}@cs.auc.dk

our encoding is *not* fully abstract with respect to weak bisimulation in the π-calculus and consider what constraints must be imposed on the π-calculus to get full abstractness. The work presented in this paper is thus related to the results achived by Sangiorgi in [San96] but arose independently. A main difference is in the choice of calculus; Sangiorgi employs the matching construct of the π-calculus, whereas the encoding presented in this paper does not and restricts its attention to the asynchronous π-calculus.

An encoding of the ς-calculus into the asynchronous π-calculus is interesting from several points of view. Firstly, most object oriented languages works with pointers, whereas the ς-calculus uses explicit substitution. In the π-calculus the basic entity is names that can be thought of as pointers; thus an encoding of the ς-calculus into the π-calculus shows how to use pointers to represent substitution. Secondly, the encoding presented in this paper also hints at the possibility of using a model checker for the π-calculus to verify properties of ς-calculus expressions. Thirdly, there is the implementation issue. When implementing programming languages on distributed systems, asynchronous communication is usually considered more easy to implement, so the encoding into an asynchronous calculus will also give an idea of how to implement the ς-calculus or a language based on the ς-calculus in a distributed setting. Finally, the π-calculus has also been put forward as a possible theoretical model of object-oriented programming languages, so an encoding will provide some basic insight into how one expresses object oriented features in the π-calculus.

The structure of the rest of the paper is as follows: Section 2 introduces the ς-calculus and explains its semantics. Section 3 gives the syntax and semantics of the asynchronous π-calculus that we shall use as our target calculus. In Section 4 we present our encoding of the ς-calculus into the asynchronous π-calculus and discuss alternative encodings. Section 5 and Section 6 regards operational correspondence of the encoding and relation between equivalences of ς-calculus terms and their encodings. Section 7 summarizes our results and relate them to other existing work.

2 The ς-calculus

The version of the ς-calculus we use is essentially the simple untyped object calculus of [AC96]. Objects in the ς-calculus are given by:

$$
\begin{array}{rcll}
a & ::= & [l_i = \varsigma(x_i)b_i] & \text{objects} \\
& | & x & \text{self variables} \\
& | & a.l & \text{method activation} \\
& | & a.l \Leftarrow \varsigma(x)b & \text{method override}
\end{array}
$$

Here $x_i \in \mathbf{SVar}$ range over self variables and $l_i \in \mathbf{MNames}$ range over method names. We let $\mathsf{m}(a)$ denote the set of method names and $\mathsf{fv}(a)$ the set of free self variables in a.

We give the semantics for objects as a reduction semantics.

Let $a = [l_i = \varsigma(x_i)b_i]$ with $l_i \in L$, then:

$$\begin{array}{lll} a.l_k & \rightsquigarrow \ b_k\{a/x_k\} & l_k \in L \\ a.l \Leftarrow \varsigma(x)b & \rightsquigarrow \ [l_i = \varsigma(x_i)b_i,\, l = \varsigma(x)b] & l_i \in L \setminus \{l\} \end{array}$$

The activation of the method l_k results in the method body being activated with the self variable being bound to the original object. Method override results in an object with the overridden method exchanged with the new method. In the original version of the ς-calculus it is not allowed to add methods to an object, a restriction introduced to keep the theory simple. Since the ability to add methods does not interfere with our encoding, we shall in the present paper allow addition of methods.

A context $C[\cdot]$ is an 'incomplete' ς-calculus term, and we write $C[a]$ to denote that it has been completed using the term a. The syntax of contexts is given by:

$$C[\cdot] ::= C[\cdot].l \mid C[\cdot].l \Leftarrow \varsigma(x)b \mid [\cdot]$$

Our final transition rule specifies the reduction strategy, which, given our syntax of contexts, implies leftmost reductions:

$$\frac{a \rightsquigarrow a'}{C[a] \rightsquigarrow C[a']}$$

Leftmost reduction implies that we in a term $a.l$ or $a.l \Leftarrow \varsigma(x)b$ always reduce a to an object before activating or overriding a method. That is, objects are the values of this version of the ς-calculus.

To give an intuition of how the ς-calculus works, we shall present a few simple examples (taken from [AC96])

$$\begin{array}{llll} \text{let } a & = & [l = \varsigma(x)x.l] & \text{then } \ a.l \rightsquigarrow x.l\{a/x\} = a.l \rightsquigarrow \cdots \\ \text{let } a' & = & [l = \varsigma(x)x] & \text{then } \ a'.l \rightsquigarrow x\{a'/x\} = a' \\ \text{let } a'' & = & [l = \varsigma(y)y.l \Leftarrow \varsigma(x)x] & \text{then } \ a''.l \rightsquigarrow a''.l \Leftarrow \varsigma(x)x \rightsquigarrow a'' \end{array}$$

The object a shows how we can get infinite behaviour through the use of self variables. The object a'' shows how an object can modify itself by performing a method override on a self variable.

3 The asynchronous π-calculus

The π-calculus [PW92] has previously shown its ability to encode both the λ-calculus [Mil92] and certain object oriented languages [Wal95, San96]. In [Wal95] Walker encoded a variant of the programming language POOL [Ame89] into the π-calculus and in [San96] Sangiorgi investigates another encoding of the ς-calculus into the π-calculus. Sangiorgi shows how to encode the ς-calculus in a type-correct way into an extended version of the π-calculus with a special *case* construct, which basically amounts to an extended matching operator.

As the target calculus for our translation we instead go for as simple a version of the π-calculus as possible. The version that we shall use is the asynchronous π-calculus [CS96, Bou92, HT91, HK95].

The syntax of asynchronous π-calculus is in the present paper given by:

$$P ::= \bar{a}\tilde{b} \mid P|P \mid (\nu\tilde{a})P \mid G \mid !G \qquad\qquad G ::= \mathbf{0} \mid a(\tilde{b}).P \mid \tau.P \mid G + G$$

We let $a, b, \ldots \in$ **Names** range over an infinite c ountable set of names, $\tilde{b}$ denote tuples of names, a tuple of the names a, b and c will be written as $\langle a, b, c\rangle$. $P \in$ **Proc** range over processes, $G \in$ **GProc** range over guarded processes and $Q \in$ **Guard** over guards. The use of guards ensures that we only replicate and sum guarded expressions.

We let $\mathsf{bn}(P)$, $\mathsf{fn}(P)$ and $\mathsf{n}(P)$, resp., denote the sest of bound names, free names and names of the agent P.

The difference between the synchronous and asynchronous π-calculus lies in the output construct. In the asynchronous π-calculus output is *non-blocking*, this is seen in the syntax as the absence of output prefixing ($\bar{a}b.P$). Instead, output is modelled as the parallel composition of an output atom and the process ($\bar{a}b \mid P$). The use of asynchronous communication has several advantages. Most importantly, several of the versions of bisimulation equivalence, that are distinct in the synchronous setting, coincide in an asynchronous setting. As a consequence, the algebraic theory becomes simpler [HK95, CS96].

For simplicity we shall only present the semantics of the monadic π-calculus, but it is easily extended to handle the polyadic case. The semantics is given by a labelled transition system with labels:

$$\alpha ::= \bar{x}y \mid \bar{x}(y) \mid xy$$

$\bar{x}y$ denotes the output of the free name y on the name x. Transmission of bound names uses *bound output* $\bar{x}(y)$, indicating that we transmit the bound name y over the channel x. The last label xy denotes the input of the name

y over the name x. Table 1 shows the inference rules for the asynchronous π-calculus, with the symmetric versions of [sync], [sync$_{\text{ex}}$], [comp] and [sum] omitted. We shall identify alpha-convertible terms, i.e. up to bound names.

$[\tau]$	$\dfrac{\cdot}{\tau.P \xrightarrow{\tau} P}$	$[\nu]$	$\dfrac{P \xrightarrow{\alpha} P' \quad a \notin \mathsf{n}(\alpha)}{(\nu a)P \xrightarrow{\alpha} (\nu a)P'}$
[out]	$\dfrac{\cdot}{\bar{a}b \xrightarrow{\bar{a}b} \mathbf{0}}$	[out$_{\text{ex}}$]	$\dfrac{P \xrightarrow{\bar{a}b} P' \quad a \neq b}{(\nu b)P \xrightarrow{\bar{a}(b)} P'}$
[in]	$\dfrac{\cdot}{a(b).P \xrightarrow{ac} P\{c/b\}}$	[comp]	$\dfrac{P \xrightarrow{\alpha} P' \quad \mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset}{P \vert Q \xrightarrow{\alpha} P' \vert Q}$
[sum]	$\dfrac{G \xrightarrow{\alpha} P}{G + G' \xrightarrow{\alpha} P}$	[sync]	$\dfrac{P \xrightarrow{\bar{a}b} P' \quad Q \xrightarrow{ab} Q'}{P \vert Q \xrightarrow{\tau} P' \vert Q'}$
[rep]	$\dfrac{G \xrightarrow{\alpha} P}{!G \xrightarrow{\alpha} P \vert !G}$	[sync$_{\text{ex}}$]	$\dfrac{P \xrightarrow{\bar{a}(b)} P' \quad Q \xrightarrow{ab} Q' \quad b \notin \mathsf{fn}(Q)}{P \vert Q \xrightarrow{\tau} (\nu b)(P' \vert Q')}$

Table 1: The inference rules for the asynchronous π-calculus.

Just as in the synchronous π-calculus several definitions of bisimulation exist (see [CS96] for an overview). In the present paper we shall adopt the following definition, due to Amadio, Castellani and Sangiorgi [CS96]:

Definition 1 (Asynchronous bisimulation) *A symmetric relation $\mathcal{R}$ is an asynchronous bisimulation if for all $P \mathcal{R} Q$, whenever:*

- *$P \xrightarrow{\alpha} P'$ and α is not an input, then $Q \xrightarrow{\alpha} Q'$ and $P' \mathcal{R} Q'$.*
- *$P \xrightarrow{ab} P'$ then either $Q \xrightarrow{ab} Q'$ and $P' \mathcal{R} Q'$ or $Q \xrightarrow{\tau} Q'$ and $P' \mathcal{R} (Q' \vert \bar{a}b)$.*

We write $P \sim Q$ if $P \mathcal{R} Q$ for some asynchronous bisimulaton $\mathcal{R}$.

The definition of asynchronous bisimulation is as the standard defintion for bisimulation except for the last part of the input case. This part expresses that if a process absorbs what it has just emitted, this can be "absorbed" in an internal action. A example of this (from [CS96]) is the following equational law:

$$a(b).(\bar{a}b \mid P) + \tau.P \sim \tau.P \quad b \notin \mathsf{fn}(P)$$

Weak bisimulation ($\approx$) is defined the usual way, by exchanging $\xrightarrow{\alpha}$ in the above definition with the corresponding weak transition $\overset{\alpha}{\Longrightarrow}$.

Our primary motivation for choosing the asynchronous π-calculus instead of the synchronous version is minimalism; we want to use as simple a version of the π-calculus as possible. But the choice gives us some important advantages. Most importantly, the notion of bisimilarity is unique and yields a congruence, a result that does not hold in the synchronous case or if we add matching. Also, the equational theory is simpler – in the synchronous case matching is needed to give an equational theory, whereas this is not necessary in the asynchronous π-calculus.

We use the notation $P \xrightarrow{\alpha}_d Q$ to indicate that $P \xrightarrow{\alpha} Q$ is the *unique* transition possible from P. To enhance readability we also omit the restrictions of names that no longer appear in a process, for instance, $(\nu o)(\bar{x}y)$ will be written as $\bar{x}y$ (formally, the two expressions are bisimlar).

4 Coding up the ς-calculus

The intuition behind our encoding is that the encoding of an object can be used via its value channel. More precisely, as soon as an object reduces to a value, an object reference is emitted on the value channel. An object reference is then used to activate methods by sending a value name, an object reference, and a method name to the object; the value name tells where we expect the result to be returned, the method name is the name of the method that we want to activate, the object reference that we pass is used if the called method wants to activate other methods. This is necessary in the case where a method has been overridden.

The polyadic π-calculus allows a straightforward typing discipline, called *sorting*, which ensures that suitable names are always communicated. Our encoding obeys the sorting:

$$\begin{array}{lll} \text{Method names} & l : \text{Method} & \rightarrow \ (\text{ObjRef}) \\ \text{Values} & v : \text{Value} & \rightarrow \ (\text{ObjRef}) \\ \text{Object references} & o : \text{ObjRef} & \rightarrow \ (\text{Method}, \text{ObjRef}, \text{Value}) \\ \text{Self variables} & x : \text{Var} & \end{array}$$

where $v : \text{Value} \rightarrow (\text{ObjRef})$ expresses that v ranges over the set of value channels and we only transmit object references over value channels.

We assume that the sets of method names and self variables in the above sorting coincide with the set of method names and self variables in the ς-calculus.

4.1 The encoding

The encoding $[\![\cdot]\!]_v$ presented below is parametrized with a value name v denoting the reference of the encoded object. The notation $m(-).P$ denotes $m(x).P$ for some name $x \notin \mathsf{fn}(P)$.

$$\begin{array}{rcl}
[\![a.l]\!]_v & = & (\nu v')([\![a]\!]_{v'} \mid v'(o).\bar{o}\langle l, o, v\rangle) \\
[\![[l_i = \varsigma(x_i)b_i]]\!]_v & = & (\nu o)(\bar{v}o \mid !o(l', o', v').(\bar{l}'o' \mid \sum l_i(x_i)[\![b_i]\!]_{v'})) \\
[\![x]\!]_v & = & \bar{v}x \\
[\![a.l \Leftarrow \varsigma(x)b]\!]_v & = & (\nu v')([\![a]\!]_{v'} \mid v'(o).(\nu o')(\bar{v}o' \mid !o'(l'', o'', v''). \\
& & (\bar{l}''o'' \mid l(x).[\![b]\!]_{v''} + \sum\limits_{m \in \mathsf{m}(a)\setminus\{l\}} m(-).\bar{o}\langle l'', o'', v''\rangle)))
\end{array}$$

Observe that the above encoding prohibits free self variables, since the sorting prohibits the transmission of self variables over value names.

In the following we shall usually omit the index sets when they are obvious from the context.

4.2 Notational conventions

To enhance the readability of encoded terms, we shall in the following use the abbreviation:

$$o := [l_i = \varsigma(x_i)b_i] \;=\; !o(l', o', v').(\bar{l}'o' \mid \sum l_i(x_i).[\![b_i]\!]_{v'})$$

The intuition behind this 'relay' construct $o := [l_i = \varsigma(x_i)b_i]$ is that the object $[l_i = \varsigma(x_i)b_i]$ resides at o. In other words, the object can have its methods activated by transmitting a method name, an object reference and a value name over the name o.

With this construct we can write the encoding $[\![[l_i = \varsigma(x_i)b_i]]\!]_v$ as $(\nu o)(\bar{v}o \mid o := [l_i = \varsigma(x_i)b_i])$.

As can be seen from the encoding of method activation and override, we select the actual method to be invoked by means of communication over method names. This implies that the encoding might be messed up by π-calculus contexts that use method names incorrectly and cause erroneous transitions. For instance:

$$\begin{array}{rl}
& l(-) \mid [\![[l = \varsigma(x)x].l]\!]_v \\
= & l(-) \mid (\nu v')((\nu o)(\bar{v}'o \mid o := [l = \varsigma(x)x]) \mid v'o'.\bar{o}'\langle l, o', v\rangle) \\
\xrightarrow{\tau} & l(-) \mid (\nu o)(o := [l = \varsigma(x)x] \mid \bar{o}\langle l, o', v\rangle)
\end{array}$$

$$\xrightarrow{\tau} \quad l(-) \mid new o(o := [l = \varsigma(x)x] \mid \bar{l}o \mid l(x).\bar{v}x)$$
$$\xrightarrow{\tau} \quad o := [l = \varsigma(x)x] \mid l(x).\bar{v}x$$

Observe how the input $l(-)$ consumes the message that was intended for method selection. To avoid this, we implicitly assume that the method names are restricted at the outermost level in the encoding. Furthermore observe that the encoding does not mess up things, since we always wait for the output on a restricted name in the encoding (except when choosing methods).

We have already introduced the notation $\xrightarrow{\alpha}_d$, we shall extend this notation with transitions on the form $P \xrightarrow{\tau}_{dl} Q$ indicating that the internal move from P to Q is the only possible transition except for actions on method names. And when we remember that the method names are restricted away at the outermost level and that our encoding does not create erroneous transitions, this implies that the transition is unique.

We use the notation $a \Uparrow$ to indicate that a has an infinite reduction sequence, and we write $a \Downarrow$ if a reduces to an object. In the π-calculus we use the same notation $P \Uparrow$ to indicate that P have an infinite sequence of τ-moves, and $P \Downarrow$ to indicate that P after a sequence of τ-moves will do an observable action.

4.3 Examples

To give the intuition of how the encoding works we shall consider a few examples.

Our first example is the encoding of the following simple object:

$$a = [l = \varsigma(x)x.l]$$

It only contains one method l, that when activated will activate itself indefinitely, resulting in the infinite reduction sequence $a.l \leadsto a.l \leadsto \cdots$. In our encoding this corresponds to the following behaviour:

$$\begin{array}{ll}
 & [\![a.l]\!]_v \\
= & (\nu v')((\nu o)(\bar{v}'o \mid o := [l = \varsigma(x)x.l]) \mid v'(o').\bar{o}'\langle l, o', v\rangle) \\
\xrightarrow{\tau} & (\nu o)(o := [l = \varsigma(x)x.l] \mid \bar{o}\langle l, o, v\rangle) \\
\xrightarrow{\tau} & (\nu o)(o := [l = \varsigma(x)x.l] \mid \bar{l}o \mid l(x).(\nu v')(\bar{v}'x \mid v'(o').\bar{o}'\langle l, o', v\rangle)) \\
\xrightarrow{\tau} & (\nu o)(o := [l = \varsigma(x)x.l] \mid (\nu v')(\bar{v}'o \mid v'(o').\bar{o}'\langle l, o', v\rangle)) \\
\xrightarrow{\tau} & (\nu o)(o := [l = \varsigma(x)x.l] \mid \bar{o}\langle l, o, v\rangle)
\end{array}$$

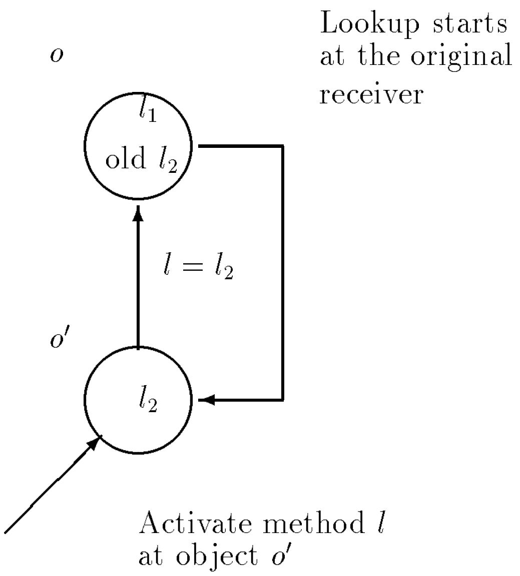

Figure 1: Method override and lookup

Our next example is somewhat more complicated and illustrates the encoding of method override. Consider the object

$$a = [l_1 = \varsigma(x)x, l_2 = \varsigma(x)b]$$

If we override a method then we get a new object with the overridden method exchanged with the overriding one. In our encoding this is simulated by generating a new "object" that handles activations of the overriding method itself and *delegates* all other method activations to the original object. This is illustrated in Figure 4.3.

The following shows how method override and lookup are handled by our encoding:

$$
\begin{array}{ll}
& [\![(a.l_2 \Leftarrow \varsigma(x)x.l_1).l_2]\!]_v \\
= & (\nu v')((\nu v'')((\nu o)(\bar{v}''o \mid o := a) \mid v''(o^*).(\nu o')(\bar{v}'o' \mid !o'(l'', o'', v'')). \\
& \quad (\bar{l}''o'' \mid l_2(x).[\![x.l_1]\!]_{v''} + l_1(-).\bar{o}^*\langle l'', o'', v''\rangle) \mid v'(o).\bar{o}\langle l_2, o, v\rangle) \\
\xrightarrow{\tau}{}^2_d & (\nu o')((\nu o)(o := [l_1 = \varsigma(x)x,\ l_2 = \varsigma(x)b] \mid !o'(l'', o'', v''). \\
& \quad (\bar{l}''o'' \mid l_2(x).[\![x.l_1]\!]_{v''} + l_1(-).\bar{o}\langle l'', o'', v''\rangle)) \mid \bar{o}'\langle l_2, o', v\rangle) \\
= & (\nu o')((\nu o)(o := [l_1 = \varsigma(x)x,\ l_2 = \varsigma(x)b] \mid !o'(l'', o'', v'').(\bar{l}''o'' \mid \\
& \quad l_2(x).(\nu v')(\bar{v}'x \mid v'(o).\bar{o}\langle l_1, o, v''\rangle) + l_1(-).\bar{o}\langle l'', o'', v''\rangle)) \mid \bar{o}'\langle l_2, o', v\rangle) \\
\xrightarrow{\tau}{}^3_{dl} & (\nu o')((\nu o)(o := [l_1 = \varsigma(x)x,\ l_2 = \varsigma(x)b] \mid !o'(l'', o'', v'').(\bar{l}''o'' \mid \\
& \quad l_2(x).(\nu v')(\bar{v}'x \mid v'(o).\bar{o}\langle l_1, o, v''\rangle) + l_1(-).\bar{o}\langle l_1, x, v''\rangle)) \mid \bar{o}'\langle l_1, o', v\rangle) \\
\xrightarrow{\tau}{}^2_{dl} & (\nu o', o)(!o(l', o', v').(\bar{l}'o' \mid l_1(x).\bar{v}'x + l_2(x).[\![b]\!]_{v'}) \mid !o'(l'', o'', v''). \\
& \quad (\bar{l}''o'' \mid l_2(x).[\![x.l_1]\!]_{v''} + l_1(-).\bar{o}\langle l'', o'', v''\rangle) \mid \bar{o}\langle l_1, o', v\rangle) \\
\xrightarrow{\tau}{}^2_{dl} & (\nu o')((\nu o)(o := [l_1 = \varsigma(x)x,\ l_2 = \varsigma(x)b] \mid !o'(l'', o'', v''). \\
& \quad (\bar{l}''o'' \mid l_2(x).[\![x.l_1]\!]_{v''} + l_1(-).\bar{o}\langle l'', o'', v''\rangle)) \mid \bar{v}o')
\end{array}
$$

Observe how the object reference of the *original* receiver is passed on when the receiver does not have the requested method itself. This ensures that when the correct method is found, we know where to start looking for other methods.

4.4 Alternative encodings

In [San96] (and a previous version of this paper) a somewhat different encoding of the untyped ς-calculus is presented. The difference is in the use of matching instead of choice and communication:

$$
\begin{array}{rcl}
[\![a.l]\!]_v & = & (\nu v')([\![a]\!]_{v'} \mid v'(o).\bar{o}\langle l, o, v\rangle) \\
[\![[l_i = \varsigma(x_i)b_i]]\!] & = & (\nu o)(\bar{v}(o) \mid !o(l', o', v').(\prod[l' = l_i](\nu x_i)(\bar{x}_i o' \mid [\![b_i]\!]_{v'}))) \\
[\![x]\!]_v & = & x(o).\bar{v}o \\
[\![a.l \Leftarrow \varsigma(x)b]\!] & = & (\nu v')([\![a]\!]_{v'} \mid v'(o).(\nu o')(\bar{v}o' \mid !o'(l'', o'', v''). \\
& & \quad ([l'' = l](\nu x)(\bar{x}o'' \mid [\![b]\!]_{v''}) \mid [l'' \neq l]\bar{o}\langle l'', o'', v''\rangle)))
\end{array}
$$

Other possibilities exist: In [San96] the typed ς-calculus is translated into an extended version of the ς-calculus with a special case construct instead of matching. This encoding does not use the 'relay' construct when dealing with method override. Instead the original object is setd a special message with the name of the overriding method, which it then inserts instead of the original method. This implies that each method will need two method names, one for method activation and one for method override.

5 Operational Correspondence

In this section we present number of results about our encoding, which together show the soundness of our encoding w.r.t. the operational semantics of the two calculi.

Our first lemma is a substitution lemma. It states that the 'relay' construct in parallel with an encoded object corresponds to binding an object reference to a self variable within the object.

Lemma 1 *Let* $a = [l_i = \varsigma(x_i)b_i]$*, then*

$$(\nu o)(o := a \mid [\![b]\!]_v\{o/x\}) \sim [\![b\{a/x\}]\!]_v$$

Proof. Induction in the structure of b.

$b = y$: We have two cases, either $y = x$ and we have:

$$\begin{array}{rcl}(\nu o)(o := a \mid [\![x]\!]_v\{o/x\}) & = & (\nu o)(o := a \mid \bar{v}o) \\ & = & [\![a]\!]_v \\ & = & [\![x\{a/x\}]\!]_v\end{array}$$

or $y \neq x$ and:

$$\begin{array}{rcl}(\nu o)(o := a \mid [\![y]\!]_v\{o/x\}) & = & (\nu o)(o := a \mid \bar{v}y) \\ & \sim & [\![y]\!]_v \\ & = & [\![y\{a/x\}]\!]_v\end{array}$$

$b = b'.l$: Assume w.l.o.g. that $v' \notin \mathsf{fn}(o := a)$, then

$$\begin{array}{rl} & (\nu o)(o := a \mid [\![b'.l]\!]_v\{o/x\}) \\ = & (\nu o)(o := a \mid (\nu v')([\![b']\!]_{v'}\{o/x\} \mid v'(o).\bar{o}\langle l, o, v\rangle)) \\ \overset{\mathsf{IH}}{\sim} & (\nu v')([\![b'\{a/x\}]\!]_{v'} \mid v'(o).\bar{o}\langle l, o, v\rangle) \\ = & [\![b'\{a/x\}.l]\!]_v \\ = & [\![b.l\{a/x\}]\!]_v\end{array}$$

$b = [l_i = \varsigma(x_i)b_i]$: Assume w.l.o.g. that $x_i \neq x$, then

$$
\begin{aligned}
& (\nu o)(o := a \mid [\![[l_i = \varsigma(x_i)b_i]]\!]_v\{o/x\}) \\
= \;& (\nu o)(o := a \mid (\nu o')(\bar{v}o' \mid !o'(l'', o'', v'')(\bar{l}''o'' \mid \sum l_i(x_i).[\![b_i]\!]_{v''}))) \\
\sim \;& (\nu o')(\bar{v}o') \mid !o'(l'', o'', v'').(\bar{l}''o'' \mid \qquad (1) \\
& \quad \sum l_i(x_i).(\nu o)(o := a \mid [\![b_i]\!]_{v''}))) \\
\overset{\mathsf{IH}}{\sim} \;& (\nu o')(\bar{v}o' \mid !o'(l'', o'', v'').(\bar{l}''o'' \mid \sum l_i(x_i).[\![b_i\{a/x\}]\!]_{v''})) \\
= \;& [\![[l_i = \varsigma(x_i)b_i\{a/x\}]]\!]_v \\
= \;& [\![[l_i = \varsigma(x_i)b_i]\{a/x\}]\!]_v
\end{aligned}
$$

In (1) we distribute the replication over the sum. It can be shown that this is valid if every free occurrence of o in each component is a negative subject, that is o is only used for output.

$b = b'.l \Leftarrow \varsigma(y)c$: Assume w.l.o.g. that $y \neq x$, then

$$
\begin{aligned}
& (\nu o)(o := a \mid [\![b'.l \Leftarrow \varsigma(y)c]\!]_v\{o/x\}) \\
= \;& (\nu o)(o := a \mid (\nu v')([\![b']\!]_{v'} \mid v'(o^*).(\nu o')(\bar{v}o' \mid !o'(l'', o'', v''). \\
& \quad (\bar{l}''o'' \mid l(y).[\![c]\!]_{v''} + \sum m(-).\bar{o}^*\langle l'', o'', v''\rangle))))\{o/x\}) \\
\sim \;& (\nu v')((\nu o)(o := a \mid [\![b']\!]_{v'}\{o/x\}) \mid v'(o^*).(\nu o')(\bar{v}o' \mid \qquad (2) \\
& \quad !o'(l'', o'', v'').(\bar{l}''o'' \mid l(y).(\nu o)(o := a \mid [\![c]\!]_{v''}\{o/x\}) + \\
& \quad \sum m(-).\bar{o}\langle l'', o'', v''\rangle))) \\
\overset{\mathsf{IH}}{\sim} \;& (\nu v')([\![b'\{a/x\}]\!]_{v'} \mid v'(o^*).(\nu o')(\bar{v}o' \mid !o'(l'', o'', v''). \\
& \quad (\bar{l}''o'' \mid l(y).[\![c\{a/x\}]\!]_{v''} + \sum m(-).\bar{o}^*\langle l'', o'', v''\rangle))) \\
= \;& [\![b\{a/x\}.l \Leftarrow \varsigma(y)c\{a/x\}]\!]_v \\
= \;& [\![(b.l \Leftarrow \varsigma(y)c)\{a/x\}]\!]_v
\end{aligned}
$$

The distribution of the replication in (2) sound for the same reasons as in (1).

□

As an immediate corollary we get that the encoding is sound w.r.t. the rule for method calls:

Corollary 1 *Let* $a = [l_i = \varsigma(x_i)b_i]$ *for* $l_i \in L$ *then*

$$[\![a.l_j]\!]_v \approx [\![b_j\{a/x_j\}]\!]_v \textit{ for all } l_j \in L$$

PROOF. We have:

$$\begin{array}{rcl}
[\![a.l_j]\!]_v & = & (\nu v')((\nu o)(\bar{v}'o \mid o := a) \mid v'(o').\bar{o}'\langle l_j, o', v\rangle) \\
& \xrightarrow{\tau}_d & (\nu o)(o := a \mid \bar{o}\langle l_j, o, v\rangle) \\
& \xrightarrow{\tau}_{dl} & (\nu o)(o := a \mid \bar{l}_j o \mid \sum l_i(x_i).[\![b_i]\!]_v) \\
& \xrightarrow{\tau}_d & (\nu o)(o := a \mid [\![b_j]\!]\{o/x_j\}) \\
& \sim & [\![b_j\{a/x_j\}]\!]_v
\end{array}$$

□

Using algebraic techniques from the asynchronous π-calculus, we can also show that method override is sound:

Lemma 2 *Let $a = [l_i = \varsigma(x_i)b_i]$ for $l_i \in L$, then*

$$[\![a.l \Leftarrow \varsigma(x)b]\!]_v \approx [\![[l_i = \varsigma(x_i)b_i,\ l = \varsigma(x)b]]\!]_v$$

PROOF. We have:

$$\begin{array}{rl}
& [\![a.l \Leftarrow \varsigma(x)b]\!]_v \\
= & (\nu v')([\![a]\!]_{v'} \mid v'(o').(\nu o)(\bar{v}o \mid !o(l'', o'', v''). \\
& \quad (\bar{l}''o'' \mid l(x).[\![b]\!]_{v''} + \sum m_i(-).\bar{o}'\langle l'', o'', v''\rangle)))\quad m_i \in \mathsf{m}(a) \setminus \{l\} \\
\xrightarrow{\tau}_d & (\nu o^*)(!o^*(l', o', v').(\bar{l}'o' \mid \sum l_i(x_i).[\![b_i]\!]_{v'}) \mid (\nu o)(\bar{v}o \mid \\
& \quad !o(l'', o'', v'').(\bar{l}''o'' \mid l(x).[\![b]\!]_{v''} + \sum m_i(-).\bar{o}^*\langle l'', o'', v''\rangle)))
\end{array}$$

Now we push the replication that represents the a object under the sum $\sum m(-)$ and get:

$$\begin{array}{l}
(\nu o)(\bar{v}o \mid !o(l'', o'', v'').(\bar{l}''o'' \mid l(x).[\![b]\!]_{v''} + \sum m_i(-). \\
\quad (\nu o^*)(\bar{o}^*\langle l'', o'', v''\rangle \mid !o^*(l', o', v').(\bar{l}'o' \mid \sum l_i(x_i).[\![b_i]\!]_{v'}))))
\end{array}$$

The above is valid due to the fact than whenever o^* is a private name and the overriding method is activated, this cannot happen with o^* as the self reference, that is, if the encoding of a is activated it will be through o.

Since o^* does not occur free in any of the $[\![b_i]\!]_{v'}$ this implies that we can remove the replication of o^* and replace the internal communication over o^* with an internal move:

$$\begin{array}{c}
(\nu o)(\bar{v}o \mid !o(l'', o'', v'').(\bar{l}''o'' \mid l(x).[\![b]\!]_{v''} + \\
\sum m_i(-).\tau.(\bar{l}''o'' \mid \sum l_i(x_i).[\![b_i]\!]_{v''})))
\end{array}$$

Finally we can remove the innermost sum $(\sum l_i(x_i).[\![b_i]\!]_{v''})$ and instead bind the self variables in the outermost sum:

$$
\begin{array}{ll}
\sim & (\nu o)(\bar{v}o \mid !o(l'', o'', v'').(\bar{l}''o'' \mid l(x).[\![b]\!]_{v''} + \sum m_i(x_i).\tau.\tau.[\![b_i]\!]_{v''}))) \\
\approx & (\nu o)(\bar{v}o \mid !o(l'', o'', v'').(\bar{l}''o'' \mid l(x).[\![b]\!]_{v''} + \sum m_i(x_i).[\![b_i]\!]_{v''}))) \\
= & [\![[l_i = \varsigma(x_i)b_i,\ l = \varsigma(x)b]]\!]_v
\end{array}
$$

□

With these lemmas we are now able to prove that reductions in the ς-calculuscorrespond to transition sequences in the π-calculus. More precisely, when an object a can do a reduction and become a', then our encoding can match that by doing a series of internal actions, resulting in a π-calculus process being weakly bisimilar to a'.

Theorem 1 *If $a \rightsquigarrow b$ then $[\![a]\!]_v \xrightarrow{\tau}{}^{*} P \approx [\![b]\!]_v$.*

Proof. Induction in the structure of a.

$a = x$: We have $a \not\rightsquigarrow$.

$a = [l_i = \varsigma(x_i)b_i]$: Again $a \not\rightsquigarrow$.

$a = a'.l$: If $a \rightsquigarrow b$ it can be for two reasons:

i. $a' \rightsquigarrow a''$ and $[\![a]\!]_v = (\nu v')([\![a']\!]_{v'} \mid v'(o).\bar{o}\langle l, o, v\rangle)$. By induction there exist a P such that $[\![a']\!]_{v'} \xrightarrow{\tau}{}^{*} P \approx [\![a'']\!]_{v'}$. And then:

$$(\nu v')(P \mid v'(o).\bar{o}\langle l, o, v\rangle) \approx (\nu v')([\![a'']\!]_{v'} \mid v'(o).\bar{o}\langle l, o, v\rangle) = [\![a''.l]\!]_v$$

ii. $a' = [l_i = \varsigma(x_i)b_i]$ with $l = l_j$, $b = b_j\{^{a'}/_{x_j}\}$. We have

$$
\begin{array}{lll}
[\![a]\!]_v & = & (\nu v')((\nu o)(\bar{v}'o \mid o := [l_i = \varsigma(x_i)x_i]) \mid v'(o').\bar{o'}\langle l, o', v\rangle) \\
& \xrightarrow{\tau}_d & (\nu o)(o := [l_i = \varsigma(x_i)x_i] \mid \bar{o}\langle l, o, v\rangle) \\
& \xrightarrow{\tau}{}^{2}_{dl} & \underbrace{(\nu o)(o := [l_i = \varsigma(x_i)x_i] \mid [\![b_j]\!]_v\{^{o}/_{x_j}\})}_{P}
\end{array}
$$

According to Lemma 1 we have $P \sim [\![b_j\{^{[l_i = \varsigma(x_i)b_i]}/_{x_j}\}]\!]_v$.

$a = a'.l \Leftarrow \varsigma(x)c$: Again we have two cases.

i. $a' \rightsquigarrow a''$ is handled as in the previous case.

ii. $a' = [l_i = \varsigma(x_i)b_i]$ and $b = [l_i = \varsigma(x_i)b_i,\ l = \varsigma(x)c]$. According to Lemma 2 we have:

$$[\![[l_i = \varsigma(x_i)b_i].l \Leftarrow \varsigma(x)c]\!]_v \approx [\![[l_i = \varsigma(x_i)b_i, l = \varsigma(x)c]]\!]_v$$

$\square$

Our encoding signals the reduction of ς-calculus term to an object as the output of an object identifier, this we express as:

Theorem 2 *If* $a \rightsquigarrow^* [l_i = \varsigma(x_i)b_i] = b$ *then* $[\![a]\!]_v \xrightarrow{\tau}{}^*_{dl} \xrightarrow{\bar{v}(o)} \approx o := b$.

Proof. We use induction in the length of $a \rightsquigarrow^n a'$.

Basis $n = 0$**:** We must have $a = [l_i = \varsigma(x_i)b_i]$ and for the encoding we have

$$\begin{array}{rcl} [\![[l_i = \varsigma(x_i)b_i]]\!]_v & = & (\nu o)(\bar{v}o \mid o := [l_i = \varsigma(x_i)b_i]) \\ & \xrightarrow{\bar{v}(o)} & o := [l_i = \varsigma(x_i)b_i] \end{array}$$

Induction step: Assume that the theorem holds for sequences of reduction steps of length n. We now consider a reduction sequence of length $n+1$, that is we have

$$a \rightsquigarrow a' \rightsquigarrow^n b = [l_i = \varsigma(x_i)b_i]$$

According to the induction hypothesis there exists a P such that:

$$[\![a']\!]_v \xrightarrow{\tau}{}^*_{dl} \xrightarrow{\bar{v}(o)} P \approx o := b$$

According to Theorem 1 there exist a Q such that $[\![a]\!]_v \xrightarrow{\tau}{}^* Q \approx [\![a']\!]_v$. We now have the following sequence:

$$[\![a]\!]_v \xrightarrow{\tau}{}^*_{dl} Q \approx [\![a']\!]_v \xrightarrow{\tau}{}^*_{dl} \xrightarrow{\bar{v}(o)} P \approx o := b$$

Since $Q \approx [\![a']\!]_v$ the must exist a Q' such that

$$[\![a]\!]_v \xrightarrow{\tau}{}^* Q \xrightarrow{\tau}{}^* \xrightarrow{\bar{v}(o)} Q' \approx P \approx o := b$$

$\square$

The relationship between transitions in the π-calculus encoding and reductions in the ς-calculus is somewhat more difficult to express, since the π-calculus encoding may need to do some internal computation before being ready to simulate an object. We relate reductions through the output of an object identifier. If we after a series of internal actions see an external action, then this is because the original ς-calculus term can reduce to an object.

Theorem 3 *If* $\mathsf{fv}(a) = \emptyset$ *and* $[\![a]\!]_v \xrightarrow{\tau}{}^* \xrightarrow{\alpha} P$ *then* $\alpha = \bar{v}(o)$ *and*

$$a \rightsquigarrow^* [l_i = \varsigma(x_i)b_i] = b \text{ and } P \approx o := b$$

Proof. We shall use induction in the number of τ-moves prior to an observable action.

Basis $n = 0$: The only process immediately capable of performing an observable action is:

$$[\![[l_x = \varsigma(x_x)i_x]]\!]_v \xrightarrow{\bar{v}(o)} o := [l_x = \varsigma(x_x)i_x]$$

And obviously the theorem holds. The other "possibility" $[\![x]\!]_v$ is prohibited.

Induction step: Assume that the theorem holds for τ-sequences of length n. We now consider at τ-sequence of length $n+1$ before an external action.

For $[\![a]\!]_v$ to have any τ-moves a must be either:

$a = a'.l$: Here we have

$$[\![a'.l]\!]_v = (\nu v')([\![a']\!]_{v'} \mid v'(o).\bar{o}\langle l, o, v\rangle) \xrightarrow{\tau}{}^{n+1} P$$

Now $[\![a']\!]_{v'}$ must have less than $n+1$ τ-moves before an observable action (remember, we disregard observable actions from method activations), since the only action possible action is with v' as subject, resulting in *one* internal communication in $[\![a]\!]_v$.

Therefore, by induction

$$[\![a']\!]_{v'} \xrightarrow{\tau}{}^m_{dl} \xrightarrow{\bar{v}'(o')} Q \approx o' := c \quad (m \leq n)$$

and combining things we get

$$(\nu v')([\![a']\!]_{v'} \mid v'(o).\bar{o}\langle l, o, v\rangle) \xrightarrow{\tau}{}^{m+1}_{dl} (\nu o')(o' := c \mid \bar{o}'\langle l, o', v\rangle)$$

Applying the induction hypothesis once more

$$(\nu o')(o' := c \mid \bar{o}'\langle l, o', v\rangle) \xrightarrow{\tau}{}^k \xrightarrow{\bar{v}(o)} \approx o := b \quad (k = n - m)$$

All in all

$$(\nu v')([\![a']\!]_{v'} \mid v'(o).\bar{o}\langle l, o, v\rangle) \xrightarrow{\tau}{}^m \xrightarrow{\tau} \xrightarrow{\tau}{}^k \xrightarrow{\bar{v}(o)} \approx o := b$$

$a = a'.l \Leftarrow \varsigma(x)c$: In the encoding we have:

$$[\![a'.l \Leftarrow \varsigma(x)c]\!]_v \;=\; (\nu v')([\![a']\!]_{v'} \mid v'(o).(\nu o')(\bar{v}o' \mid !o'(l'', o'', v''). \\ (\bar{l}''o'' \mid l(x).[\![c]\!]_{v''} + m(-).\bar{o}\langle l'', o'', v''\rangle)))$$

Using the same line of reasoning as in the previous case we get

$$\begin{aligned} &(\nu v')([\![a']\!]_{v'} \mid v'(o).(\nu o')(\bar{v}(o') \mid !o'(l'', o'', v''). \\ &\quad (\bar{l}''o'' \mid l(x).[\![c]\!]_{v''} + m(-).\bar{o}\langle l'', o'', v''\rangle))) \\ &\xrightarrow{\tau}{}^{n+1} \xrightarrow{\bar{v}(o)} \\ &(\nu o')(o' := c' \mid !o'(l'', o'', v'').(\bar{l}''o'' \mid l(x).[\![c]\!]_{v''} + \\ &\quad m(-).\bar{o}\langle l'', o'', v''\rangle)) \end{aligned} \tag{3}$$

With $c' \approx [l_i = \varsigma(x_i)b_i]$. Now according to Lemma 2 (4) is weakly bisimilar to $o' := [l_i = \varsigma(x_i)b_i,\ l = \varsigma(x)c]$.

□

6 Equivalences for the ς-calculus

As an operational equivalence for the ς-calculus we shall use context equivalence as defined in [GR95], except that we do not take the type of contexts into account.

Definition 2 (Context equivalence) *A relation $\mathcal{R}$ is a context ς-equivalence if it is symmetric and $a\ \mathcal{R}\ b$ implies:*

- *If $a \Downarrow$ then $b \Downarrow$ and for all contexts $C[\cdot]$ we $C[a]\ \mathcal{R}\ C[b]$.*

Two objects are context ς-equivalent (written $a \simeq b$) if $a\ \mathcal{R}\ b$ for some context ς-equivalence.

In [GR95] Gordon and Rees show that context equivalence correspond to bisimulation in a labelled transition system for ς-calculus terms.

The omission of type information has important implications. When we restrict our attention to well-typed ς-calculus terms, we prohibit terms such as $[].l$, that is, objects where we try to activate nonexisting methods. It is quite easy to see that the only terms bisimilar are objects which also terminate with the attempt to activate a nonexisting method.

The use of restriction of method names at the outermost level has the implication that the expected result, namely that weak bisimulation between encoded terms implies context equivalence, does *not* hold. The problem is that once we have restricted the method names away, we cannot activate any methods and see how the encoding of the objects behaves. To illustrate the problem, consider the following two objects, that are definitely *not* context equivalent:

$$a = [l = \varsigma(x)x] \qquad b = [l = \varsigma(x)x.l]$$

With the restriction of method names at the outermost level we have:

$$\begin{array}{ll} & [\![a]\!]_v \\ = & (\nu l)((\nu o)(\bar{v}o \mid !o(o', l', v').(\bar{l}'o' \mid l(x).\bar{v}'x)) \\ \xrightarrow{\bar{v}(o)} & (\nu l)(!o(l', o', v').(\bar{l}'o' \mid l(x).\bar{v}'x)) \\ \sim & !o(l', o', v').(\bar{l}'o' \mid (\nu l)(l(x).\bar{v}'x))) \\ \sim & !o(l', o', v').\bar{l}'o' \end{array}$$

This is also the case for $[\![b]\!]_v$ so we have $[\![a]\!]_v \approx [\![b]\!]_v$ but $a.l \Downarrow$ and $b.l \Uparrow$.

On the other hand, if we drop the restriction of method names, then it is easily shown that weak bisimulation between encodings of terms implies context equivalence of the original terms.

Theorem 4 *If* $[\![a]\!]_v \approx [\![b]\!]_v$ *then* $a \simeq b$

Proof. Assume that $[\![a]\!]_v \approx [\![b]\!]_v$, then because $\approx$ is a congruence we have $C_\pi[[\![a]\!]_v] \approx C_\pi[[\![b]\!]_v]$ for all π-calculus contexts, and in particular for contexts that are encodings of some ς-calculus context. If $C_\pi[\cdot]$ is the encoding of the ς-calculus context $C[\cdot]$ then we have $C_\pi[[\![a]\!]_v] = [\![C[a]]\!]_{v'}$ and $C_\pi[[\![b]\!]_v] = [\![C[b]]\!]_{v'}$ for some v'.

Now we know from Theorem 3 that if $[\![C[a]]\!]_{v'}$ performs an observable action on v', then it is because $C[a]$ terminates. Furthermore, if $[\![C[a]]\!]_{v'} \approx [\![C[b]]\!]_{v'}$ then $[\![C[b]]\!]_{v'}$ then must also have an observable action on v', what implies that $C[b]$ must also terminate, and vice versa.

That is, if we have $[\![a]\!]_v \approx [\![b]\!]_v$ then we have $a \simeq b$. □

The reverse implication of Theorem 4 does not hold; for instance

$$a = [l = \varsigma(x)x] \quad b = [l = \varsigma(x)a]$$

If we do not allow addition of methods in method override, then we have $a \simeq b$, but their encodings are easily distinguished. For the first we have the following transition sequence:

$$[\![a]\!]_v \xrightarrow{\bar{v}(o)} o := a \xrightarrow{o(l,o,v)} \xrightarrow{\tau}{}^{*}_{dl} \xrightarrow{\bar{v}(o)} o := a$$

And for the encoding of the second:

$$[\![b]\!]_v \xrightarrow{\bar{v}(o)} o := b \xrightarrow{o(l,o,v)} \xrightarrow{\tau}{}^{*}_{dl} \xrightarrow{\bar{v}(o')} o := b \mid o' := a$$

If we allow addition of methods, then a and b are not context bisimilar, since b after the activation of l will "lose" the added method.

It is not enough to allow the addition of methods, to see why consider:

$$a = [l = \varsigma(x)x] \qquad b = [l = \varsigma(x)x.l \Leftarrow \varsigma(x)x]$$

These two object are congruent, even if we allow addition of methods, but their encodings are not weak bisimilar since the overriding in b results in the creation of a *new* object reference.

If fact, when we remove the restriction of method names we obtain a very fine-grained equivalence between ς-calculus terms. This is essentially due to the fact two objects are weakly bisimilar in their encoding when they activate the same methods at the same time. For instance, consider:

$$a = [l = \varsigma(x)[l_1 = \varsigma(y)x].l_1] \quad b = [l = \varsigma(x)[l_2 = \varsigma(y)x].l_2]$$

These two object are obviously equivalent with respect to their reductions, but since the innermost methods have different names the encodings of a and b are not weakly bisimilar.

To characterize ς-equivalence, we need to restrict ourselves to π-calculus contexts which are encodings of ς-calculus contexts:

Definition 3 *A symmetric relation $\mathcal{R}_v$ is a $\varsigma\pi$-bisimulation if $P \mathrel{\mathcal{R}_v} Q$ implies that if $P \xrightarrow{\tau}{}^{*}_{dl} \xrightarrow{\alpha} P'$ then $Q \xrightarrow{\tau}{}^{*}_{dl} \xrightarrow{\alpha} Q'$, $\alpha = \bar{v}(o)$ and*

- *For all $l \in Method$:*

$$(\nu o)(P' \mid \bar{o}\langle l, o, v\rangle) \mathrel{\mathcal{R}_v} (\nu o)(Q' \mid \bar{o}\langle l, o, v\rangle)$$

- *For all $l \in Method$, x and c:*

$$(\nu o, o')(P' \mid \bar{v}(o').!o'(l'', o'', v'').(\bar{l}''o'' \mid l(x).[\![c]\!]_{v''} + \sum m(x).\bar{o}\langle l'', o'', v''\rangle))$$
$$\mathcal{R}_v$$
$$(\nu o, o')(Q' \mid \bar{v}(o').!o'(l'', o'', v'').(\bar{l}''o'' \mid l(x).[\![c]\!]_{v''} + \sum m(x).\bar{o}\langle l'', o'', v''\rangle))$$

Using the operational correspondence between ς-calculus terms and their encoding we can prove that if two ς-calculus terms are ς-equivalent then their encodings are contained in some $\varsigma\pi$-bisimulation.

Theorem 5 *If $a \simeq b$, then there exist a $\varsigma\pi$-bisimulation $\mathcal{R}_v$ such that $[\![a]\!]_v \, \mathcal{R}_v \, [\![b]\!]_v$.*

Proof. Let $\mathcal{R}_v = \{([\![c_1]\!]_v, [\![c_2]\!]_v) \mid c_1 \simeq c_2\}$. Obviously we have $[\![a]\!]_v \, \mathcal{R}_v \, [\![b]\!]_v$. We now claim that $\mathcal{R}_v$ is a $\varsigma\pi$-bisimulation up to $\approx$.

Let $[\![c_1]\!]_v \, \mathcal{R}_v \, [\![c_2]\!]_v$, we now consider what behaviour $[\![c_1]\!]_v$ can have.

$[\![c_1]\!]_v \Uparrow$: By the operational correspondence this must be because $c_1 \Uparrow$. Since $c_1 \simeq c_2$, $[\![c_2]\!]_v$ must also diverge.

$[\![c_1]\!]_v \xrightarrow{\tau}{}^* P \not\rightarrow$: This implies that $c_1 \rightsquigarrow^* C[[l_i = \varsigma(x_i)b_i].l]$ with $l_i \in L_1$ and $l \notin L_1$. Since $c_1 \simeq c_2$ we also have $c_2 \rightsquigarrow^* C[[l_j = \varsigma(x_j)b_j].l]$ with $l_j \in L_2$ and $l' \notin L_2$ and again by the operational correspondence we have $[\![c_2]\!]_v \xrightarrow{\tau}{}^* Q \not\rightarrow$.

$[\![c_1]\!]_v \xrightarrow{\tau} \xrightarrow{\bar{v}(o)} P$: This implies that $a \rightsquigarrow^* [l_i = \varsigma(x_i)b_i]$ with $P \approx o := [l_i = \varsigma(x_i)b_i]$. Since $c_1 \simeq c_2$, $c_2 \rightsquigarrow [l_j = \varsigma(x_j)b_j]$ with $[l_i = \varsigma(x_i)b_i] \simeq [l_j = \varsigma(x_j)b_j]$ and therefore $[\![c_2]\!]_v \xrightarrow{\tau} \xrightarrow{\bar{v}(o)} Q$ with $Q \approx o := [l_j = \varsigma(x_j)b_j]$.

By the definition of $\simeq$ we must have $[l_i = \varsigma(x_i)b_i].l \simeq [l_j = \varsigma(x_j)b_j].l$ and $[l_i = \varsigma(x_i)b_i].l \Leftarrow \varsigma(c)x \simeq [l_j = \varsigma(x_j)b_j].l \Leftarrow \varsigma(c)x$. Now, using Lemma 1, we have:

$$\begin{array}{rl} & (\nu o)(P \mid \bar{o}(l,o,v)) \approx [\![[l_i = \varsigma(x_i)b_i].l]\!]_v \\ \mathcal{R}_v & [\![[l_j = \varsigma(x_j)b_j].l]\!]_v \approx (\nu o)(Q \mid \bar{o}(l,o,v)) \end{array}$$

and Lemma 2

$$\begin{array}{rl} & (\nu o, o')(P \mid \bar{v}(o').!o'(l'', o'', v'').(\bar{l}''o'' \mid l(x).[\![c]\!]_{v''} + \\ & \quad \sum m(-).\bar{o}\langle l'', o'', v''\rangle)) \\ \approx & [\![[l_i = \varsigma(x_i)b_i,\ l = \varsigma(x)c]]\!]_v \\ \mathcal{R}_v & [\![[l_j = \varsigma(x_j)b_j,\ l = \varsigma(x)c]]\!]_v \\ \approx & (\nu o)(Q \mid \bar{o}(l,o,v)) \end{array}$$

□

The reverse implication also holds; that is, if we can find a $\varsigma\pi$-bisimulation for the encoding of two ς-calculus terms, then they are ς-equivalent.

Theorem 6 *If $[\![a]\!]_v \, \mathcal{R}_v \, [\![b]\!]_v$ for some $\varsigma\pi$-bisimulation then $a \simeq b$.*

Proof. Let $[\![a]\!]_v \; \mathcal{R}_v \; [\![b]\!]_v$ for some $\varsigma\pi$-bisimulation. We now claim that a and b have the same termination behaviour in all contexts.

Clearly, by the operational correspondence, if $[\![a]\!]_v \Uparrow$ then also $a \Uparrow$ and the same holds for b.

Also by the operational correspondence, if $[\![a]\!]_v \xrightarrow{\tau} \xrightarrow{\bar{v}(o)} P$ then $a \leadsto^* [l_i = \varsigma(x_i)b_i]$ with $P \approx o := [l_i = \varsigma(x_i)b_i]$ and similarly for b. Therefore a and b must have the same termination behaviour. By definition of $\mathcal{R}_v$ this must also hold for all context that a and b can be put in. □

7 Conclusions and further work

In this paper we have described how to encode the simple untyped object calculus of Abadi and Cardelli into the asynchronous π-calculus without matching. We chose this calculus to see how simple a calculus we would need to encode the ς-calculus. As this paper shows, it is possible to encode the ς-calculus into our target calculus, but the price is somewhat high. The proofs of operational correspondence rely on specific assumptions about the use of method names in the encoding, and as Section 6 shows, weak bisimilarity between encoded terms gives us a very fine-grained equivalence between ς-calculus terms, since it requires two objects to have the same method activation behaviour to be equivalent.

In [San96] Sangiorgi investigates well-typed encodings of the ς-calculus into the π-calculus. His work is very similar to the work presented in this paper ; however, Sangiorgi uses a version of the synchronous π-calculus extended with a case operator.

The authors are currently working on an encoding of the imperative object calculus [AC95b, AC95a]. The imperative object calculus is interesting in that it incorporates references to objects, a phenomenon common to many object-oriented programming languages. Because of the presence of references, the semantics of the imperative object calculus is quite similar to our encoding.

References

[Abr89] S. Abramsky. The lazy lambda-calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1989.

[AC94a] Martin Abadi and Luca Cardelli. A semantics of object types. In *Proceedings of the 9th IEEE Symposium on Logics in Computer*

Science, pages 332–341. IEEE Computer Society, Computer Society Press, 1994.

[AC94b] Martin Abadi and Luca Cardelli. A theory of primitive objects: Second-order systems. In *Proceedings of European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 1–25. Springer-Verlag, 1994.

[AC95a] M. Abadi and L. Cardelli. An imperative object calculus: Basic typing and soundness. In *SIPL '95 - Proc. Second ACM SIGPLAN Workshop on State in Programming Languages.* Technical Report UIUCDCS-R-95-1900, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.

[AC95b] Martin Abadi and Luca Cardelli. An imperative object calculus. *Theory and Practice of Object Systems*, 1(3):151–166, 1995.

[AC96] Martin Abadi and Luca Cardelli. A theory of primitive objects – untyped and first-order systems. *Information and Computation*, 125(2):78–102, 1996.

[Ame89] P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):396–411, 1989.

[Bou92] Gérard Boudol. Asynchrony and the π-calculus. Technical report, INRIA Sophia-Antipolis, 1992.

[CS96] Roberto M. Amadio Ilaria Castellani and Davide Sangiorgi. On bisimulations for the asynchronous π-calculus. In *Proceedings of CONCUR 96*, Lecture Notes in Computer Science. Springer-Verlag, 1996.

[GR95] A. D. Gordon and G. D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, January 1996.* ACM, 1995.

[HK95] Martin Hansen Hans Hüttel and Josva Kleist. Bisimulations for asynchronous mobile processes. In *Proceedings of the Tbilisi Symposium on Language, Logic and Computation*, 1995.

[HT91] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *ECOOP 91*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer-Verlag, 1991.

[Mil92] Robin Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[PW92] Robin Milner Joachim Parrow and David Walker. A calculus of mobile processes — part I and II. *Information and Computation*, 100:1–77, 1992.

[San96] Davide Sangiorgi. An interpretation of typed objects into typed π-calculus. Research Report RR-3000, INRIA Sophia Antipolis, August 1996.

[Wal95] David Walker. Objects in the π-calculus. *Information and Computation*, 116:253–271, 1995.

Recent Publications in the BRICS Report Series

RS-96-38 **Hans Hüttel and Josva Kleist.** ***Objects as Mobile Processes*. October 1996. 23 pp.**

RS-96-37 **Gerth Stølting Brodal and Chris Okasaki.** ***Optimal Purely Functional Priority Queues*. October 1996. 27 pp. To appear in *Journal of Functional Programming*, 6(6), December 1996.**

RS-96-36 **Luca Aceto, Willem Jan Fokkink, and Anna Ingólfsdóttir.** ***On a Question of A. Salomaa: The Equational Theory of Regular Expressions over a Singleton Alphabet is not Finitely Based*. October 1996. 16 pp.**

RS-96-35 **Gian Luca Cattani and Glynn Winskel.** ***Presheaf Models for Concurrency*. October 1996. 16 pp. Presented at the *Annual Conference of the European Association for Computer Science Logic*, CSL '96.**

RS-96-34 **John Hatcliff and Olivier Danvy.** ***A Computational Formalization for Partial Evaluation (Extended Version)*. October 1996. To appear in *Mathematical Structures in Computer Science*.**

RS-96-33 **Jonathan F. Buss, Gudmund Skovbjerg Frandsen, and Jeffrey Outlaw Shallit.** ***The Computational Complexity of Some Problems of Linear Algebra*. September 1996. 39 pp.**

RS-96-32 **P. S. Thiagarajan.** ***Regular Trace Event Structures*. September 1996. 34 pp.**

RS-96-31 **Ian Stark.** ***Names, Equations, Relations: Practical Ways to Reason about 'new'*. September 1996. ii+22 pp.**

RS-96-30 **Arne Andersson, Peter Bro Miltersen, and Mikkel Thorup.** ***Fusion Trees can be Implemented with AC^0 Instructions only*. September 1996. 8 pp.**

RS-96-29 **Lars Arge.** ***The I/O-Complexity of Ordered Binary-Decision Diagram Manipulation*. August 1996. 35 pp. An extended abstract version appears in Staples, Eades, Kato, and Moffat, editors, *Algorithms and Computation: 6th International Symposium*, ISAAC '95 Proceedings, LNCS 1004, 1995, pages 82–91.**